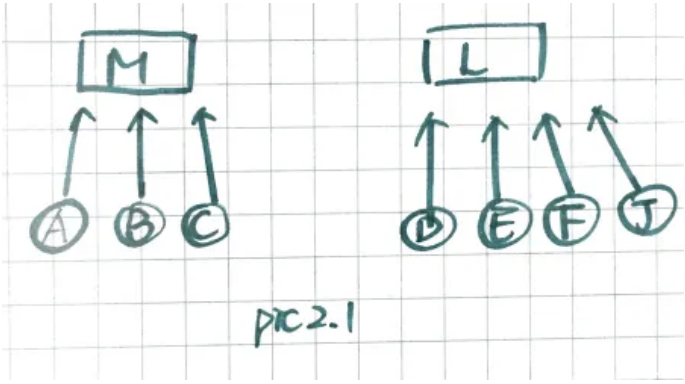
Stomachache007

喷萝卜牛杂味儿的香水，擦煎饼果子味儿的面霜

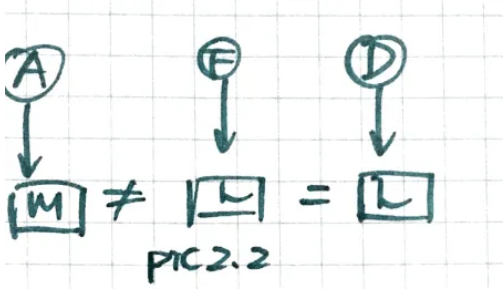
九章算法高级班笔记2.数据结构（上）

并查集 Union Find

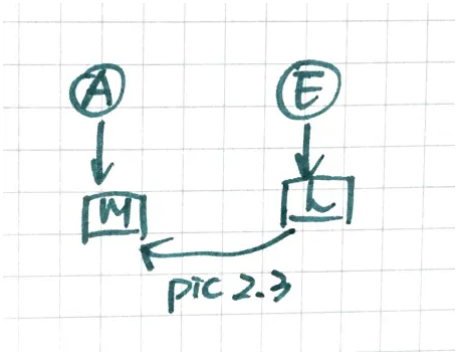
并查集是这样一种数据机构:
A, B, C三个人在Microsoft工作, 给他们三个人每个人都安个指针指向Microsoft. 这个指针相对于你工作单位的腰牌,代表你所属的阵营, 说明你是这个阵营的小弟.
D, E, F, J四个人在Linkedin工作, 给这四个人每个人都安个腰牌指针指向大哥Linkedin.



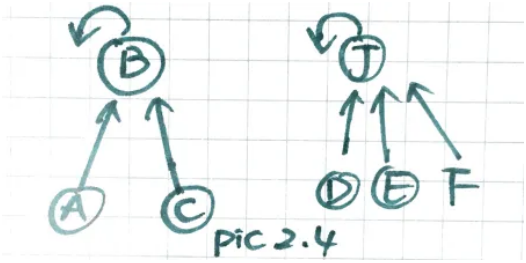
这个指针腰牌有神马用呢? 就是假设A在饭桌上遇到了F, A一看对放腰牌是L公司, F一看对方腰牌是M公司, 就都知道对方是不是自己一家公司的. 又来个人D, D的腰牌是L公司, 这下F和D也立即就知道他俩是同事.



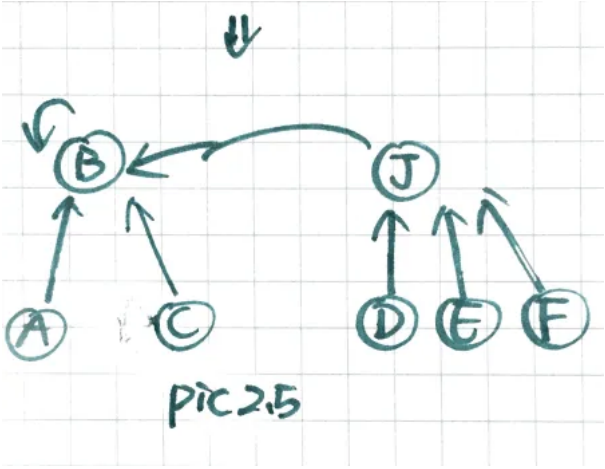
这时候, 发生了一件事儿, 就是M公司Microsoft把L公司Linkedin收购了. 但是Linkedin也不小, 底下一撮子的小弟, 一个个改腰牌好麻烦. 然后公司想了个办法, D, E, F, J的大哥不变, 依然由Linkedin担当, 但是Linkedin要百Microsoft当大哥. 这样饭桌上即使聊起来, A可以看下E的腰牌, 噢, 你大哥是L啊, 是我大哥M手下L管的小弟, 咱俩还是一个阵营哒~



这里, 我们可以不用M和L这个公司名字, 直接在公司的所有员工里, 选个权利最大的做代表, 比如M公司的A, B, C我就选了B (Bill Gates), L公司呢选J, 大家的大哥就分别是这两个人:



这时候呢, 如果公司之间合并收购, 就让J拜B当大哥, 大家就是一伙儿的啦



并查集的精髓即三个操作：

1. 初始化

初始化后每一个元素的父亲节点是它本身，即自己是自己的大哥（也可以根据情况而变），举个栗子:

```
private int[] father = null;
    public Initialization(int n) {
        // initialize your data structure here.
        father = new int[n + 1];
        for (int i = 1; i <= n; ++i)
            father[i] = i;
    }
```

2. 查找

查找一个元素所在的集合，其精髓是找到这个元素的大哥！这个才是并查集判断和合并的最终依据。
判断两个元素是否属于同一集合，只要看他们是不是最终都归一个大哥管,模板如下:

```
 private int find(int x) {
    if (father[x] == x) {
        return x;
    }
    return father[x] = find(father[x]);
}
```

3. 合并

合并两个不相交集合就是让一伙的老大哥拜另一伙儿的老大哥为大哥,模板为:

```
public void connect(int a, int b) {
    // Write your code here
    int root_a = find(a);
    int root_b = find(b);
    if (root_a != root_b)
```

```
            father[root_a] = root_b;
        }
```

路径压缩和时间复杂度：

无路径压缩是这样的:

```
    private int find(int x) {
        if (father[x] == x) {
            return x;
        }
        return find(father[x]);
    }
```

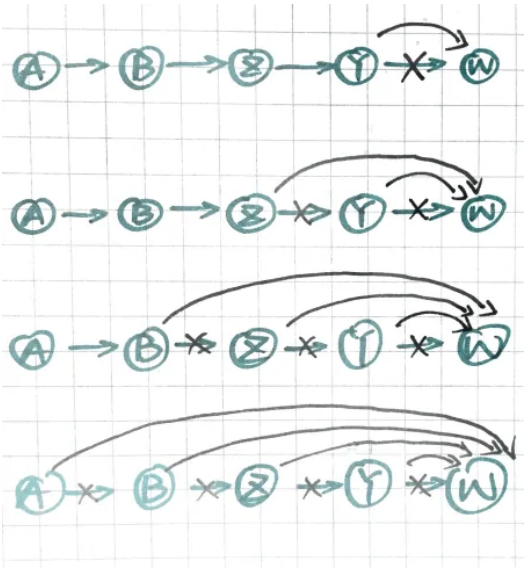

有路径压缩是

```
    private int find(int x) {
        if (father[x] == x) {
            return x;
        }
        return father[x] = find(father[x]);
    }
```

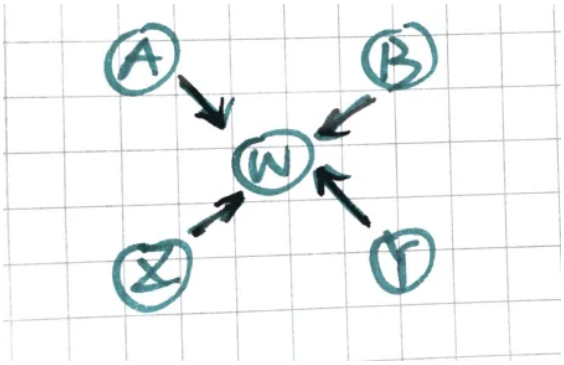
其中路径压缩是这样的原理:

假设A->B->Z->Y->W这样的路径, 我们想查询A的老大哥, 需要从A走过B, Z, Y到W, 途中经过5个点, B查询经过4个点... 每次这样好麻烦:

于是我们在递归回溯的过程中, 把每层的father大哥都变成最大的大哥:



然后我们就由链条型多步找大哥变为众星捧月型一步找大哥:



其中:

```
private int find(int x) {
    if (father[x] == x) {
        return x;
    }
    return father[x] = find(father[x]);
}
```

的时间复杂度第一次是O(n), 但是多次下来是log*n (见 https://en.wikipedia.org/wiki/Iterated_logarithm), 证明略

x	$\lg^* x$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

因为1-5都很小, 所以find基本上是O(1)的操作.
如果find是O(1), 那么union也是O(1)时间复杂度的操作.

Connecting Graph

Given n nodes in a graph labeled from 1 to n. There is no edges in the graph at beginning.

You need to support the following method:

connect(a, b), add an edge to connect node a and node b`.
query(a, b), check if two nodes are connected.

这道题很容易直接想到bfs, bfs的遍历, 时间复杂度是O(kn), 其中k是操作次数.
如果用并查集的方法, 时间复杂度每次操作都是O(1), 所以总共是O(k).

```
public class ConnectingGraph {
```

```
    private int[] father = null;

    private int find(int x) {
        if (father[x] == x) {
            return x;
        }
        return father[x] = find(father[x]);
    }

    public ConnectingGraph(int n) {
        // initialize your data structure here.
        father = new int[n + 1];
        for (int i = 1; i <= n; ++i)
            father[i] = i;
    }

    public void connect(int a, int b) {
        // Write your code here
        int root_a = find(a);
        int root_b = find(b);
        if (root_a != root_b)
            father[root_a] = root_b;
    }

    public boolean  query(int a, int b) {
        // Write your code here
        int root_a = find(a);
        int root_b = find(b);
        return root_a == root_b;
    }
}
```

Connecting Graph II

Given n nodes in a graph labeled from 1 to n. There is no edges in the graph at beginning.

You need to support the following method:

connect(a, b), an edge to connect node a and node b
query(a), Returns the number of connected component nodes which include node a.

```
public class ConnectingGraph2 {

    private int[] father = null;
    private int[] size = null;

    private int find(int x) {
        if (father[x] == x) {
            return x;
        }
        return father[x] = find(father[x]);
    }

    public ConnectingGraph2(int n) {
        // initialize your data structure here.
        father = new int[n + 1];
        size = new int[n + 1];
        for (int i = 1; i <= n; ++i) {
            father[i] = i;
            size[i] = 1;
        }
    }

    public void connect(int a, int b) {
        // Write your code here
        int root_a = find(a);
        int root_b = find(b);
        if (root_a != root_b) {
            father[root_a] = root_b;
            size[root_b] += size[root_a];
        }
    }

    public int query(int a) {
        // Write your code here
        int root_a = find(a);
        return size[root_a];
    }
}
```

Connecting Graph问题的总结

并查集原生操作:

1. 查询两个元素是否在同一个集合内
2. 合并两个元素所在的集合

并查集的派生操作：

1. 查询某个元素所在集合的元素个数
2. 查询当前集合的个数

Number of Islands

Given a boolean 2D matrix, 0 is represented as the sea, 1 is represented as the island. If two 1 is adjacent, we consider them in the same island. We only consider up/down/left/right adjacent.

Find the number of islands.
这道题, 用bfs或者dfs的时候, k个操作, 时间复杂度是O(k*m*n), 其中m是行数, n是列数.
但是其实每次操作, 我们并不需要遍历整张图, 只需要在每个点附近的局部做处理就行, 这种bfs/dfs下的局部处理, 可以想到并查集. 需要做的的事是: 把其他的操作,转化成查找和合并这两件事情.

值得注意的是, 二位矩阵转一维有个公式:
二位矩阵中的(x, y)某点如果想变为一维矩阵中的某个位置, id转化公式如下, 其中m为二维矩阵的列数:

```
 (x, y) ---> ID :      ID = x * m + y
ID ----> (x, y):      x = ID / m
                      y = ID % m
```

这道题的题解:

```
class UnionFind {

    private int[] father = null;
    private int count;

    private int find(int x) {
        if (father[x] == x) {
            return x;
        }
        return father[x] = find(father[x]);
    }

    public UnionFind(int n) {
        // initialize your data structure here.
        father = new int[n];
        for (int i = 0; i < n; ++i) {
            father[i] = i;
        }
    }

    public void connect(int a, int b) {
        // Write your code here
        int root_a = find(a);
        int root_b = find(b);
        if (root_a != root_b) {
            father[root_a] = root_b;
            count --;
        }
    }

    public int query() {
        // Write your code here
        return count;
    }

    public void set_count(int total) {
        count = total;
    }
}

public class Solution {
    /**
     * @param grid a boolean 2D matrix
     * @return an integer
     */
    public int numIslands(boolean[][] grid) {
        int count = 0;
        int n = grid.length;
```

```
        if (n == 0)
            return 0;
        int m = grid[0].length;
        if (m == 0)
            return 0;
        UnionFind union_find = new UnionFind(n * m);

        int total = 0;
        for(int i = 0;i < grid.length; ++i)
            for(int j = 0;j < grid[0].length; ++j)
            if (grid[i][j])
                total ++;

        union_find.set_count(total);
        for(int i = 0;i < grid.length; ++i)
            for(int j = 0;j < grid[0].length; ++j) {
             if (grid[i][j]) {
                if (i > 0 && grid[i - 1][j]) {
                    union_find.connect(i * m + j, (i - 1) * m + j);
                }
                if (i  0 && grid[i][j - 1]) {
                    union_find.connect(i * m + j, i * m + j - 1);
                }
                if (j < m - 1 && grid[i][j + 1]) {
                    union_find.connect(i * m + j, i * m + j + 1);
                }
            }
        return union_find.query();
    }
}
```

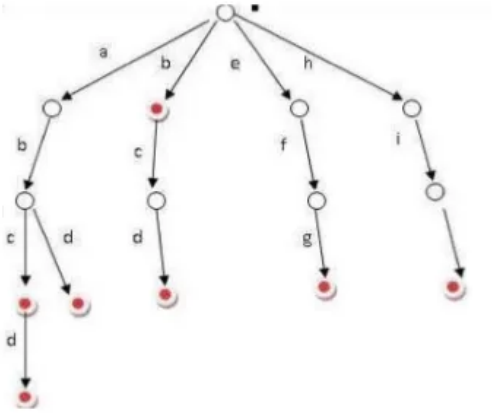

此道题类似于把大象装冰箱, 一共分三步:

1. 初始化
2. 检查双下左右四个方向
3. 合并岛屿减count

   时间复杂度为: O(m*N + 4k)*, 其中k为操作的次数, 小于*bfs/dfs的O(mN*k)

Trie树

Implement Trie



Trie树的三步走：

1. 建立根的须 char
2. 须实体化 hashmap
3. 记录结尾 hasWord

C++的implementation：

1. 装trie的指针的数组 Trie *children[26]
2. 记录结尾 is_end
3. 构造函数复制1和2

```
struct Trie {
    Trie *children[26];
    bool is_end;
    Trie() {
        // sizeof(children)/ sizeof(children[0])
        for (int i = 0; i < (sizeof(children)/ sizeof(Trie *)); i++) {
            this->children[i] = NULL;
        }
        this->is_end = false;
    }
};

class WordDictionary {
public:
    /*
     * @param word: Adds a word into the data structure.
     * @return: nothing
     */

    Trie *root = new Trie;
```

```
class TrieNode {
    // Initialize your data structure here.
    char c;
    HashMap children = new HashMap();
    boolean hasWord;

    public TrieNode(){

    }

    public TrieNode(char c){
        this.c = c;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode cur = root;
        HashMap curChildren = root.children;
        char[] wordArray = word.toCharArray();
        for(int i = 0; i < wordArray.length; i++){
            char wc = wordArray[i];
            if(curChildren.containsKey(wc)){
                cur = curChildren.get(wc);
            } else {
                TrieNode newNode = new TrieNode(wc);
                curChildren.put(wc, newNode);
                cur = newNode;
            }
            curChildren = cur.children;
            if(i == wordArray.length - 1){
                cur.hasWord= true;
            }
        }
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
```

```
        if(searchWordNodePos(word) == null){
            return false;
        } else if(searchWordNodePos(word).hasWord)
          return true;
          else return false;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        if(searchWordNodePos(prefix) == null){
            return false;
        } else return true;
    }

    public TrieNode searchWordNodePos(String s){
        HashMap children = root.children;
        TrieNode cur = null;
        char[] sArray = s.toCharArray();
        for(int i = 0; i < sArray.length; i++){
            char c = sArray[i];
            if(children.containsKey(c)){
                cur = children.get(c);
                children = cur.children;
            } else{
                return null;
            }
        }
        return cur;
    }
}
```

Hash VS Trie

O(1)不是代表一个字符, O(1)代表整个字符串

	Hash	Trie
构建	O(n)	O(n)
查询	O(1)	O(1)

对于 a, aa, aaa, aaaa的情况

```
          Hash                  Trie
存储      10个a                 5个a节点
可用操作   有/无查询             有/无/前缀查询
代码量    1行                   75~100行
```

所以选择hash原因是代码量小, 但是涉及到前缀查询的时候, 考虑trie树

Add and Search Word

Design a data structure that supports the following two operations:
addWord(word) and search(word)

search(word) can search a literal word or a regular expression string containing only letters a-z or .

A . means it can represent any one letter.

如果用dfs在矩阵里面一个一个字符遍历, 时间复杂度至少为O(m*n* * *m*n).
dfs+trie树就只需要遍历在trie树里面有前缀的, 所以会快
注意不能用.作为trie树的第27叉.

```
class TrieNode {

    public TrieNode[] children;
    public boolean hasWord;

    public TrieNode() {
        children = new TrieNode[26];
        for (int i = 0; i < 26; ++i)
            children[i] = null;
        hasWord = false;
    }
}
```

```
public class WordDictionary {
    private TrieNode root;

    public WordDictionary(){
        root = new TrieNode();
    }

    // Adds a word into the data structure.
    public void addWord(String word) {
        // Write your code here
        TrieNode now = root;
        for(int i = 0; i < word.length(); i++) {
            Character c = word.charAt(i);
            if (now.children[c - 'a'] == null) {
                now.children[c - 'a'] = new TrieNode();
            }
            now = now.children[c - 'a'];
        }
        now.hasWord = true;
    }

    boolean find(String word, int index, TrieNode now) {
        if(index == word.length()) {
            return now.hasWord;
        }

        Character c = word.charAt(index);
        if (c == '.') {
            for(int i = 0; i < 26; ++i)
            if (now.children[i] != null) {
                if (find(word, index+1, now.children[i]))
                    return true;
            }
            return false;
        } else if (now.children[c - 'a'] != null) {
            return find(word, index+1, now.children[c - 'a']);
        } else {
            return false;
        }
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    public boolean search(String word) {
        // Write your code here
        return find(word, 0, root);
    }
```

```
}
```


C++的实现：

```
struct Trie {
    Trie *children[26];
    bool is_end;
    Trie() {
        // sizeof(children)/ sizeof(children[0])
        for (int i = 0; i < (sizeof(children)/ sizeof(Trie *)); i++) {
            this->children[i] = NULL;
        }
        this->is_end = false;
    }
};

class WordDictionary {
public:
    Trie *root = new Trie;
    void addWord(string &word) {
        // write your code here
        Trie *r = root;
        for (int i = 0; i < word.size(); i++) {
            char c = word[i];
            int  pos = c - 'a';
            if (r->children[pos] == NULL) {
                r->children[pos] = new Trie;
            }
            r = r->children[pos];
        }
        r->is_end = 1;
    }

    bool search(string &word) {
        // write your code here
       return find(word, 0, root);
    }
    bool find(string &word, int index, Trie* r) {
        if (index >= word.size()) {
            if (r->is_end == 1) {
                return true;
            }
            return false;
        }
        if (root == NULL) {
            return false;
        }

        char c = word[index];
        if (c != '.') {
            int pos = c - 'a';
            if (r->children[pos] == NULL) {
                return false;
            }
            return find(word, index+1, r->children[pos]);
        } else {
            int found = 0;
            for (int i = 0; i < 26; i++) {
                if (!found && r->children[i] != NULL) {
                    found = find(word, index+1, r->children[i]);
                }
            }
            return found;
        }

    }
};
```

Word Search II

Given a matrix of lower alphabets and a dictionary. Find all words in the dictionary that can be found in the matrix. A word can start from any position in the matrix and go left/right/up/down to the adjacent position.

三步走:

1. 字典建成trie树
2. 用dfs的方法遍历矩阵,同时在Trie上搜索前缀是否存在
3. 查询所有Trie里面有可能出现的字符

注意:

1. trie树建立要选空间占地面积小的字典, 而不是矩阵, 因为如果用矩阵的话, 还要dfs遍历矩阵, 太麻烦
2. 要有一个查询走没走过的board[ ][ ]

Notice

- There are at least 1 and at most 1000 words.
- All words will have the exact same length.
- Word length is at least 1 and at most 5.
- Each word contains only lowercase English alphabet `a-z`.

可以问面试官的clarify的问题:

1. size
2. length
3. lowercase / uppercase

```
public class Solution {
    /**
     * @param board: A list of lists of character
     * @param words: A list of string
```

```
     * @return: A list of string
     */

    class TrieNode {
        String s;
         boolean isString;
         HashMap subtree;
         public TrieNode() {
            // TODO Auto-generated constructor stub
             isString = false;
             subtree = new HashMap();
             s = "";
         }
    };


    class TrieTree{
        TrieNode root ;
        public TrieTree(TrieNode TrieNode) {
            root = TrieNode;
        }
        public void insert(String s) {
            TrieNode now = root;
            for (int i = 0; i < s.length(); i++) {
                if (!now.subtree.containsKey(s.charAt(i))) {
                    now.subtree.put(s.charAt(i), new TrieNode());
                }
                now  =  now.subtree.get(s.charAt(i));
            }
            now.s = s;
            now.isString  = true;
        }
        public boolean find(String s){
            TrieNode now = root;
            for (int i = 0; i < s.length(); i++) {
                if (!now.subtree.containsKey(s.charAt(i))) {
                    return false;
                }
                now  =  now.subtree.get(s.charAt(i));
            }
            return now.isString ;
        }
    };

    public int []dx = {1, 0, -1, 0};
    public int []dy = {0, 1, 0, -1};

    public void search(char[][] board, int x, int y, TrieNode root, List
ans) {
```

```
        if(root.isString == true)
        {
            // -------这行很重要 因为每个结束都是靠它下面一层递归加string
进结果
            // 而它下面的递归可能进去很多次，并且dad这种palindrome本来也
会重复-----------------------
            if(!ans.contains(root.s)){
                ans.add(root.s);
            }
        }
        if(x = board.length || y = board[0].length || board[x][y]==0 ||
root == null)
            return ;
        if(root.subtree.containsKey(board[x][y])){
            for(int i = 0; i < 4; i++){
                char now = board[x][y];
                board[x][y] = 0;
                search(board, x+dx[i], y+dy[i], root.subtree.get(now),
ans);
                board[x][y] = now;
            }
        }

    }

    public List wordSearchII(char[][] board, List words) {
        List ans = new ArrayList();

        TrieTree tree = new TrieTree(new TrieNode());
        for(String word : words){
            tree.insert(word);
        }
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[i].length; j++){
                search(board, i, j, tree.root, ans);
            }
        }
        return ans;
        // write your code here

    }

}
```

Word Squares

Given a set of words without duplicates, find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 ≤ k < max(numRows, numColumns).

For example, the word sequence [“ball”,“area”,“lead”,“lady”] forms a word square because each word reads the same both horizontally and vertically.

```
 b a l l
a r e a
l e a d
l a d y
```

假设单词平均长度为m, 单词个数为n, 用简单的dfs我们需要的时间复杂度是A(m*n), A代表排列.但是不要气馁, 写程序不是一步登天, 我们先想蠢办法, 然后一步一步优化. 杨过的路子不好走,咱们学郭靖, 一点点来.

因为有前缀相关的, 所以想到trie树, 总共主要有两步骤:

1. 找前缀
2. 验证能否放进去

```
public class Solution {
    class TrieNode {
```

```
    List startWith;
    TrieNode[] children;

    TrieNode() {
        startWith = new ArrayList();
        children = new TrieNode[26];
    }
}

class Trie {
    TrieNode root;

    Trie(String[] words) {
        root = new TrieNode();
        for (String w : words) {
            TrieNode cur = root;
            for (char ch : w.toCharArray()) {
                int idx = ch - 'a';
                if (cur.children[idx] == null)
                    cur.children[idx] = new TrieNode();
                cur.children[idx].startWith.add(w);
                cur = cur.children[idx];
            }
        }
    }

    List findByPrefix(String prefix) {
        List ans = new ArrayList();
        TrieNode cur = root;
        for (char ch : prefix.toCharArray()) {
            int idx = ch - 'a';
            if (cur.children[idx] == null)
                return ans;

            cur = cur.children[idx];
        }
        ans.addAll(cur.startWith);
        return ans;
    }
}

public List wordSquares(String[] words) {
    List ans = new ArrayList();
    if (words == null || words.length == 0)
        return ans;
    int len = words[0].length();
    Trie trie = new Trie(words);
    List ansBuilder = new ArrayList();
    for (String w : words) {
```

```
                ansBuilder.add(w);
                search(len, trie, ans, ansBuilder);
                ansBuilder.remove(ansBuilder.size() - 1);
            }

            return ans;
        }

        private void search(int len, Trie tr, List ans,
                List ansBuilder) {
            if (ansBuilder.size() == len) {
                ans.add(new ArrayList(ansBuilder));
                return;
            }

            int idx = ansBuilder.size();
            StringBuilder prefixBuilder = new StringBuilder();
            for (String s : ansBuilder)
                prefixBuilder.append(s.charAt(idx));
            List startWith = tr.findByPrefix(prefixBuilder.toString());
            for (String sw : startWith) {
                ansBuilder.add(sw);
                search(len, tr, ans, ansBuilder);
                ansBuilder.remove(ansBuilder.size() - 1);
            }
        }
    }
```


这个在搜索type ahead上面经常使用到.

Segment Tree

线段树是一个二叉树结构， 它能做所有heap能做的操作，并且可以logn时间查找到某个区间的最大和最小值

```
      Heap        Segment Tree
push    O(logn)         O(logn)
pop     O(logn)         O(logn)
top     O(1)            O(1)
modify   X              O(logn)
```

线段树的三个操作

1. Query
2. Build
3. Modify

线段树的性质：

```
1. 二叉树
2. Parent 区间被平分
  Leftson = 左区间
  Rightson = 右区间
  叶子节点不可分
3. 每个节点表示区间（叶子节点代表一个元素）
```

Segment Tree Query

For an integer array (index from 0 to n-1, where n is the size of this array), in the corresponding SegmentTree, each node stores an extra attribute max to denote the maximum number in the interval of the array (index from start to end).

Design a query method with three parameters root, start and end, find the maximum number in the interval [start, end] by the given root of segment tree.

Notice

It is much easier to understand this problem if you finished Segment Tree Build first.
Example
For array [1, 4, 2, 3], the corresponding Segment Tree is:

```
                [0, 3, max=4]
               /             \
        [0,1,max=4]        [2,3,max=3]
        /         \        /         \
[0,0,max=1] [1,1,max=4] [2,2,max=2], [3,3,max=3]
```


query(root, 1, 1), return 4

query(root, 1, 2), return 4

query(root, 2, 3), return 3

query(root, 0, 2), return 4

节点区间和要查找区间的关系分四种情况：
1.节点区间包含查找区间->查找区间递归向下
2.节点区间不相交于查找区间 ->查找区间停止搜索
3.节点区间相交不包含于查找区间->查找区间分裂成两段区间，一段于被节点区间包含，另一段不相交
4. 节点区间相等于查找区间-> 返回值查找的结果

查询最多LOGN层，时间复杂度为O(LOGN)

```
public class Solution {
    /**
     *@param root, start, end: The root of segment tree and
     *                         an segment / interval
     *@return: The maximum number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        // write your code here
        if(start == root.start && root.end == end) { // 相等
            return root.max;
        }


        int mid = (root.start + root.end)/2;
        int leftmax = Integer.MIN_VALUE, rightmax = Integer.MIN_VALUE;
        // 左子区
        if(start <= mid) {
            if( mid < end) { // 分裂
                leftmax =  query(root.left, start, mid);
            } else { // 包含
                leftmax = query(root.left, start, end);
            }
            // leftmax = query(root.left, start, Math.min(mid,end));
        }
        // 右子区
        if(mid < end) { // 分裂 3
            if(start <= mid) {
                rightmax = query(root.right, mid+1, end);
            } else { //  包含
                rightmax = query(root.right, start, end);
            }
            //rightmax = query(root.right, Math.max(mid+1,start), end);
        }
        // else 就是不相交
        return Math.max(leftmax, rightmax);
    }
}
```

Segment Tree Build

The structure of Segment Tree is a binary tree which each node has two attributes start and end denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

The root's start and end is given by build method.
The left child of node A has start=A.left, end=(A.left + A.right) / 2.
The right child of node A has start=(A.left + A.right) / 2 + 1, end=A.right.
if start equals to end, there will be no children for this node.
Implement a build method with two parameters start and end, so that we can create a corresponding segment tree with every node has the correct start and end value, return the root of this segment tree.

Clarification
Segment Tree (a.k.a Interval Tree) is an advanced data structure which can support queries like:

which of these intervals contain a given point
which of these points are in a given interval
See wiki:
Segment Tree
Interval Tree

Example
Given start=0, end=3. The segment tree will be:

```
            [0,  3]
          /        \
      [0,  1]     [2, 3]
```

```
        /    \       /    \
    [0, 0]  [1, 1] [2, 2] [3, 3]
```

Given start=1, end=6. The segment tree will be:

```
            [1,  6]
          /        \
       [1,  3]      [4,  6]
       /     \      /     \
    [1, 2]   [3,3] [4, 5]   [6,6]
     /  \           /   \
  [1,1]  [2,2]    [4,4] [5,5]
```

口诀：

自上而下递归分裂
自下而上回溯更新

时间复杂度为O(N)：

一共logn层， 每层的元素个数是分别是
[1, 2, 4, 8...n]

```
 O   = 1+2+4+...+n
       = (2^(logn+1) -1)/ (1 - 2)
       = 2n
       = n
```

```
public class Solution {
    /**
```

```
     *@param start, end: Denote an segment / interval
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int start, int end) {
        // write your code here
        if(start > end) {  // check core case
            return null;
        }

        SegmentTreeNode root = new SegmentTreeNode(start, end);

        if(start != end) {
            int mid = (start + end) / 2;
            root.left = build(start, mid);
            root.right = build(mid+1, end);

            // root.max = Math.max(root.left.max, root.right.max);
        }
        return root;
    }
}
```

Segment Tree Build II

The structure of Segment Tree is a binary tree which each node has two attributes start and end denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

The root's start and end is given by build method.
The left child of node A has start=A.left, end=(A.left + A.right) / 2.
The right child of node A has start=(A.left + A.right) / 2 + 1, end=A.right.
if start equals to end, there will be no children for this node.
Implement a build method with a given array, so that we can create a corresponding segment tree with every node value represent the corresponding interval max value in the array, return the root of this segment tree.
Clarification
Segment Tree (a.k.a Interval Tree) is an advanced data structure which can support queries like:

which of these intervals contain a given point
which of these points are in a given interval
See wiki:
Segment Tree
Interval Tree

Example
Given [3,2,1,4]. The segment tree will be:

```
                     [0,  3] (max = 4)
                     /             \
             [0,  1] (max = 3)         [2, 3]  (max = 4)
               /        \                 /             \
[0, 0](max = 3)  [1, 1](max = 2)  [2, 2](max = 1) [3, 3] (max = 4)
```

```
public class Solution {
    /**
     *@param A: a list of integer
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int[] A) {
        // write your code here
        return buildTree(0, A.length - 1, A);
    }

    public SegmentTreeNode buildTree(int start, int end, int[] A) {
        if (start > end)
            return null;
```

```
        if (start == end) {
            return new SegmentTreeNode(start, end, A[start]);
        }
        SegmentTreeNode node = new SegmentTreeNode(start, end, A[start]);
        int mid = (start + end) / 2;
        node.left = this.buildTree(start, mid, A);
        node.right = this.buildTree(mid + 1, end, A);
        if (node.left != null && node.left.max > node.max)
            node.max = node.left.max;
        if (node.right != null && node.right.max > node.max)
            node.max = node.right.max;
        return node;
    }
}
```

c++ implementation:

```
/**
 * Definition of SegmentTreeNode:
 * class SegmentTreeNode {
 * public:
 *     int start, end, max;
 *     SegmentTreeNode *left, *right;
 *     SegmentTreeNode(int start, int end, int max) {
 *         this->start = start;
 *         this->end = end;
 *         this->max = max;
 *         this->left = this->right = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     *@param A: a list of integer
     *@return: The root of Segment Tree
     */
    SegmentTreeNode * build(vector<int>& A) {
        // write your code here
        return buildTree(0, A.size()-1, A);
    }

    SegmentTreeNode * buildTree(int start, int end, vector<int>& A) {
        if (start > end)
            return NULL;
```

```
        SegmentTreeNode * node = new SegmentTreeNode(start, end,
A[start]);
        if (start == end)
            return node;
        int mid = (start + end) / 2;
        node->left = buildTree(start, mid, A);
        node->right = buildTree(mid+1, end, A);
        if (node->left && node->left->max > node->max)
            node->max = node->left->max;
        if (node->right && node->right->max > node->max)
            node->max = node->right->max;
        return node;
    }
};
```

Segment Tree Modify

For a Maximum Segment Tree, which each node has an extra value max to store the maximum value in this node's interval.

Implement a modify function with three parameter root, index and value to change the node's value with [start, end] = [index, index] to the new given value. Make sure after this change, every node in segment tree still has the max attribute with the correct value.

Notice

We suggest you finish problem Segment Tree Build and Segment Tree Query first.

Have you met this question in a real interview? Yes
Example
For segment tree:

```
                  [1, 4, max=3]
                /               \
```

```
    [1, 2, max=2]                [3, 4, max=3]
   /             \              /             \
[1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=3]
```

if call modify(root, 2, 4), we can get:

```
                [1, 4, max=4]
              /               \
    [1, 2, max=4]               [3, 4, max=3]
   /             \              /             \
[1, 1, max=2], [2, 2, max=4], [3, 3, max=0], [4, 4, max=3]
```

or call modify(root, 4, 0), we can get:

```
                [1, 4, max=2]
              /               \
    [1, 2, max=2]               [3, 4, max=0]
   /             \              /             \
[1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=0]
```

注意

```
1. heap无法对元素进行修改；hash heap和平衡二叉树可以
2. modify最多查找logn层，时间复杂度是logn
3. 口诀：自上而下递归查找
        自下而上回溯更新
4. 以数组下标来建立线段树
```

```
public class Solution {
    /**
     *@param root, index, value: The root of segment tree and
     *@ change the node's value with [index, index] to the new given
value
     *@return: void
```

```
     */
    public void modify(SegmentTreeNode root, int index, int value) {
        // write your code here
        if(root.start == index && root.end == index) { // 查找到
            root.max = value;
            return;
        }

        // 查询
        int mid = (root.start + root.end) / 2;
        if(root.start <= index && index <=mid) {
            modify(root.left, index, value);
        }

        if(mid < index && index <= root.end) {
            modify(root.right, index, value);
        }
        //更新
        root.max = Math.max(root.left.max, root.right.max);
    }
}
```

Interval Minimum Number

Given an integer array (index from 0 to n-1, where n is the size of this array), and an query list. Each query has two integers [start, end]. For each query, calculate the minimum number between index start and end in the given array, return the result list.

Notice

We suggest you finish problem Segment Tree Build, Segment Tree Query and Segment Tree Modify first.

Have you met this question in a real interview? Yes
Example
For array [1,2,7,8,5], and queries [(1,2),(0,4),(2,4)], return [2,1,5]

```
 class SegmentTreeNode {
    public int start, end, min;
    public SegmentTreeNode left, right;
    public SegmentTreeNode(int start, int end, int min) {
          this.start = start;
          this.end = end;
          this.min = min;
          this.left = this.right = null;
    }
}
public class Solution {
    /**
     *@param A, queries: Given an integer array and an query list
     *@return: The result list
     */
    public SegmentTreeNode build(int start, int end, int[] A) {
        // write your code here
        if(start > end) {  // check core case
            return null;
        }

        SegmentTreeNode root = new SegmentTreeNode(start, end,
Integer.MAX_VALUE);

        if(start != end) {
            int mid = (start + end) / 2;
            root.left = build(start, mid, A);
            root.right = build(mid+1, end, A);

            root.min = Math.min(root.left.min, root.right.min);
        } else {
            root.min = A[start];
        }
        return root;
    }
    public int query(SegmentTreeNode root, int start, int end) {
        // write your code here
        if(start == root.start && root.end == end) { // 相等
            return root.min;
        }



        int mid = (root.start + root.end)/2;
        int leftmin = Integer.MAX_VALUE, rightmin = Integer.MAX_VALUE;
        // 左子区
        if(start <= mid) {
            if( mid < end) { // 分裂
                leftmin =  query(root.left, start, mid);
```

```
            } else { // 包含
                leftmin = query(root.left, start, end);
            }
        }
        // 右子区
        if(mid < end) { // 分裂 3
            if(start <= mid) {
                rightmin = query(root.right, mid+1, end);
            } else { //  包含
                rightmin = query(root.right, start, end);
            }
        }
        // else 就是不相交
        return Math.min(leftmin, rightmin);
    }

    public ArrayList<Integer> intervalMinNumber(int[] A,
                                                ArrayList<Interval>
queries) {
        // write your code here
        SegmentTreeNode root = build(0, A.length - 1, A);
        ArrayList ans = new ArrayList<Integer>();
        for(Interval in : queries) {
            ans.add(query(root, in.start, in.end));
        }
        return ans;
    }
}
```

线段树小总结

线段树的三大操作

```
 1.Query    O(log(n))
2.Build    O(n)
3.Modify   O(log(n))
```

线段树的应用：

1. Sum
2. Maximum/ Minimum
3. Count

构建形式：

1. 下标作为建立区间
2. 值作为建立区间

本节小结：

1. union find (union O(1) find O(1)):用来查询两个元素在不在一个集合，合并两个元素所在集合，某个元素所在集合元素个数，当前集合个数。dfs/bfs的局部操作可用union find
2. trie (build O(n) searchO(1) 1代表整个字符串): 三要素：须，is_end, constructor。有前缀的时候想trie树
3. segment tree (push， pop和heap不支持的modify, query都是O(logn), top O(1), build O(n)): 五要素：start， end ， min， *left， *right。区间极值